ARCHI2 - Compte-rendu du TME2

Nicolas Phan

pour le 23 Février 2018

Table des matières

1 Modélisation de l'architecture matérielle

1.1 Question C1

Caractéristique des caches de données et d'instructions :

- **taille :** 1024 octets
- **mapping :** direct mapping, pas d'associativité
- **taille ligne :** 16 octets
- **profondeur de TEP :** 8 octets

La taille d'une ligne de cache est de 16 octets, autrement dit 4 mots, d'où : `icache_words` = 4.

Le nombre de lignes de cache est :

$$\text{nombre de lignes} = \frac{\text{taille totale du cache}}{\text{taille d'une ligne}} = \frac{1024}{16} = 64$$

Puis comme le mapping est direct :

$$\text{nombre d'ensembles} = \text{nombre de lignes} = 64$$

D'où `icache_sets` = 64.

Le cache est à correspondance directe, il n'y a donc qu'un niveau d'associativité. D'où `icache_ways` = 1.

1.2 Question C2

Au démarrage de la machine, la RAM est vide. Pour que la machine démarre, il faut bien qu'un premier programme s'exécute et charge l'OS de la ROM vers la RAM notamment. Ce programme, le code de boot, doit être nécéssairement dans une mémoire contenant des données au moment du démarrage, autrement dit dans la ROM.

1.3 Question C3

Contrairement aux autres segments, le segment TTY réfère à des registres qui peuvent être modifiés par un composant autre que le processeur, sans que le processeur le sache. Cacher ces registres entrainerait donc des incohérences mémoire car une donnée cached deviendait incohérente sans que le processeur le sache.

1.4 Question C4

Les segments kcode, kunc, kdata contiennent des données appartenant au noyau, elle sont donc protégées, de plus le segment tty contient des registres permettant de communiquer avec le périphérique TTY et seul l'OS a le droit de communiquer directement avec les périphériques (les programmes user doivent passer par lui s'ils veulent communiquer avec un périphérique) donc le segment TTY est protégé aussi.

La protection de ces segments est réalisée en les plaçant dans la moitié haute de l'espace d'adressage. Ce découpage de l'espace d'adressage en 2 moitiés permet de vérifier très simplement si une adresse est protégée ou non (par vérification du MSB de l'adresse)

Les fichiers `sys.bin` et `app.bin` contiennent déjà dans leurs métadonnées les adresses où charger les segments qu'ils contiennent, il n'y a donc pas à spécifier d'adresse dans l'appel au constructeur du loader pour ce type de fichiers.

tp2_top.cpp : Segmentation de l'espace adressable

```
1
2  #define SEG_REFSIZE_16B      0x10
3  #define SEG_REFSIZE_1KB      0x400
4  #define SEG_REFSIZE_4KB      0x1000
5  #define SEG_REFSIZE_16KB     0x4000
6  #define SEG_REFSIZE_64KB     0x10000
7
8  // segment definition
9  #define SEG_RESET_BASE   0xbfc00000
10 #define SEG_RESET_SIZE   SEG_REFSIZE_4KB
11
12 #define SEG_KCODE_BASE   0x80000000
13 #define SEG_KCODE_SIZE   SEG_REFSIZE_16KB
14
15 #define SEG_KDATA_BASE   0x82000000
16 #define SEG_KDATA_SIZE   SEG_REFSIZE_64KB
17
18 #define SEG_KUNC_BASE    0x81000000
19 #define SEG_KUNC_SIZE    SEG_REFSIZE_4KB
20
21 #define SEG_DATA_BASE    0x01000000
22 #define SEG_DATA_SIZE    SEG_REFSIZE_16KB
23
24 #define SEG_CODE_BASE    0x00400000
25 #define SEG_CODE_SIZE    SEG_REFSIZE_16KB
26
27 #define SEG_STACK_BASE   0x02000000
28 #define SEG_STACK_SIZE   SEG_REFSIZE_16KB
29
30 #define SEG_TTY_BASE     0x90000000
31 #define SEG_TTY_SIZE     SEG_REFSIZE_16B
```

tp2_top.cpp : Paramétrage des caches

```
1      size_t   icache_ways          = 1;            // instruction cache number of ways
2      size_t   icache_sets          = 64;           // instruction cache number of sets
3      size_t   icache_words         = 4;            // instruction cache number of words per line
4      size_t   dcache_ways          = 1;            // data cache number of ways
5      size_t   dcache_sets          = 64;           // data cache number of sets
6      size_t   dcache_words         = 4;            // data cache number of words per line
7      size_t   wbuf_depth           = 8;            // write buffer depth
```

2 Système d'exploitation : GIET

2.1 Question D1

Pour effectuer un appel système, un utilisateur doit appeler la fonction `sys_call()` en donnant en arguments le numéro de l'apppel système en question (quelle fonction appeler) et fournir les arguments en entrée de cette fonction.

Les différents appels système prennent 4 arguments au maximum mais peuvent en prendre moins, hors la fonction `sys_call()` prend toujours 4 arguments. Lorsqu'un appel système nécéssite moins de 4 arguments, il faut appeller `sys_call()` en donnant la valeur 0 pour les arguments non utilisés.

La fonction `syscall()` stocke le numéro de syscall et les arguments fournis dans des registres.

La transmission de ces informations (numéro et arguments de syscall) se fait via les registres `v0` et `a0` à `a3`.

2.2 Question D2

`__syscall_vector[]` est un tableau de 32 mots contenant les adresses des différentes fonctions appel système. C'est dans `sys_handler.h` qu'est initialisé ce tableau ainsi que les macros permettant d'accéder à une case par un nom intelligible plutôt que par un numéro brut.

`__cause_vector[]` fonctionne de la même manière sauf qu'il contient l'ensemble des adresses des fonction de traitement des exceptions, le tableau et les macros d'indexation sont définies dans `exc_handler.h`.

2.3 Question D3

Succession d'appels de fonctions :

- `proctime()` appelle la fonction `sys_call()` de `stdio.c`
- `sys_call()` appelle l'instruction assembleur "syscall"
- `_sys_handler`, une fonction assembleur de `giet.s`, est exécuteé. celle-ci va lire le tableau `__syscall_vector[]` pour obtenir l'adresse de la fonction syscall à appeler, puis appelle cette dernière (en branchant à l'adresse obtenue)
- `_proctime()` de `drivers.c` est la fonction vers laquelle `_sys_handler` a branché.

3 Génération du code binaire

3.1 Question E1

Le code boot doit accéder aux registres protégés du processeur, or seuls les programmes exécutés en mode superviseurs sont autorisés à y accéder donc le code boot doit nécessairement être en mode superviseur.

```
# initializes stack pointer
    la   $29,    seg_stack_base      # $29 <= seg_stack_base
    addiu   $29,    0x00002000      # stack size = 16 Kbytes
```

3.2 Question E2

La convention permettant au code de boot de récupérer l'adresse du/des points d'entrée dans le code applicatif est la suivante : Au début du segment KDATA se trouve une table de sceaux, un tableau contenant l'adresse de la/les fonctions qui sont des points d'entrée dans le code applicatif. Le code de boot n'a plus qu'à connaitre l'adresse de base du segment kdata et il pourra y lire les adresse des points d'entrée dans le code applicatif.

Ici, il n'y a qu'une application utilisateur à lancer, donc qu'un point d'entrée, qui se trouvera donc à l'adresse définie par la macro SEG_KDATA_BASE.

```

seg_reset_base  = 0xBFC00000;

seg_kcode_base  = 0x80000000;
seg_kunc_base   = 0x81000000;
seg_kdata_base  = 0x82000000;

seg_code_base   = 0x00400000;
seg_data_base   = 0x01000000;
seg_stack_base  = 0x02000000;

seg_tty_base    = 0x90000000;
seg_timer_base  = 0x91000000;
seg_ioc_base    = 0x92000000;
seg_dma_base    = 0x93000000;
seg_gcd_base    = 0x95000000;
seg_fb_base     = 0x96000000;
seg_icu_base    = 0x9F000000;
```

3.3 Question E3

Si les adresses définies dans ces deux fichiers ne sont pas égales, cela peut entrainer des bus error, des segmentation fault voire des écriture de données vers les mauvais péripheriques.

Par exemple, si les adresses de base de la RAM et du TTY sont inversées entre la définition logicielle et la définition matérielle, alors si un programme veut écrire dans les registres du TTY, il enverra une commande d'écriture à l'adresse de base du TTY (+ un potentiel offset), or pour le BCU, cette adresse correspond a la RAM donc ce dernier selectionnera la RAM comme cible, pour peu que l'OS soit charge en RAM vers les premières adresses, celui-ci peut se retrouver corrpompu. En fin de compte, un programme veut afficher un caractère à l'ecran et finit par corrompre le système d'exploitation.

3.4 Question E4

D'après le contenu de `sys.ld`, le segment RESET contient le code du fichier reset.s et le segment KCODE contient le code du systeme d'exploitation, en commençant par celui du fichier giet.s

3.5 Question E5

D'après le désassemblage de `sys.bin`, le segment RESET se situe entre les adresses `0xbcf00000` et `0xbcf00024`, il est donc de taille $\texttt{0x24} + 1 = 36 + 1 = 37 octets$.

De la même manière, le segment KCODE se situe entre `0x80000000` et `0x80002224` donc est de taille $\texttt{0x2224} + 1 =$ 8741 octets.

3.6 Question E6

```
1    while(1)
2    {
3        tty_puts(s);
4        tty_getc(&c);
5    }
```

3.7 Question E7

La fonction système `_tty_read()` ne contient pas de boucle d'attente : si aucun caractere n'est entré au clavier, la fonction n'attend pas et renvoie 0. En revanche, l'appel système `tty_getc()` contient une boucle d'attente maintenue tant que `_tty_read()` renvoie 0, autrement dit tant que l'utilisateur n'a rien tapé au clavier.

C'est donc la fonction utilisateur et non pas la fonction noyau qui contient la boucle d'attente.

3.8 Question E8

Le segment utilisateur CODE débute à l'adresse `0x00400000` et se termine à `0x0040134c`, il a donc une taille de $\texttt{0x134d} = 4941$ octets.

4 Exécution du code binaire sur le prototype virtuel

4.1 Question F1

la première transaction sur le bus est une transaction rafale de lecture de 4 mots a partir de l'adresse `0xbcf00000` (adresse de base du segment RESET), cela correspond à la lecture du bloc contenant la premiere instruction du code de boot dans la ROM.

Au démarrage de la machine, la première chose que fait le processeur est d'exécuter le code de boot mais comme au démarrage le cache est vide, la lecture de la première instruction entraine un miss compulsif. A cause de ce miss, le gestionnaire de cache va initier sur le bus une transaction de lecture du bloc contenant cette instruction.

C'est au cycle 10 qu'est exécutée la première instruction du code de boot. A ce cycle là, la réponse du cache d'instruction est valide et l'instruction en question est `0x3c1d0200`, ce qui correspond bien au code hexadecimal de l'instruction `lui sp, 0x200` qui est la première instruction du code de boot (selon le fichier `sys.bin.txt`).

La deuxième transaction est encore une fois une transaction rafale de lecture de 4 mots dans la ROM, mais cette fois à partir de l'adresse `0xbcf0000c`, 4 mots plus loin que pour la première instruction.

Le processeur exécute le code de boot, la premiàre transaction a ramené un bloc de 4 mots, contenant les 4 premières instructions du code de boot. Celles-ci ont ensuite été exécutées, mais quand le SP passe à l'instruction 5, il y a de nouveau un miss compulsif sur le cache d'instruction, qui va donc entrainer une lecture du deuxieme bloc du segment RESET. Ainsi de suite, il y aura miss compulsif toutes les 4 instructions.

4.2 Question F2

la première instruction du `main()` est l'instruction `0x27bdffd0` (`addiu sp, sp, -48`) située à l'adresse `0x004012dc` d'après les binaires decompiles. Et d'après la trace, c'est à partir du cycle 57 que cette instruction est exécutée.

4.3 Question F3

Une première transaction (commençant au cycle 90) lit un bloc contenant :

Cycle	Adresse	Donnée Lue	Traduction ASCII
94	0x01000078	6548200a	He
95	0x0100007c	206f6c6c	llo_

Une deuxieme transaction (commencant au cycle 114) lit le bloc suivant :

Cycle	Adresse	Donnée Lue	Traduction ASCII
116	0x01000080	0x6c726f57	Worl
117	0x01000084	0x0a202164	d !

Ainsi, c'est au cycle 94 que commence la première transaction corresponant à la lecture de la chaine `"Hello world!"`.
Missing : F4, C1(wbuf_depth), E7(why loop in user mode and not kern mode ?)